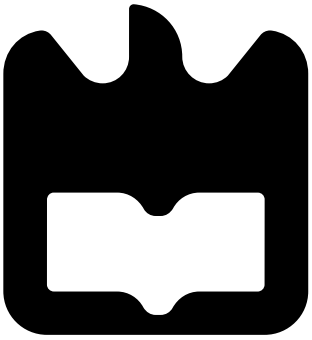
Vítor Manuel
dos Santos Marques

Desempenho de algoritmos de ordenação em *hardware* e *software* implementados em SoC

Performance of hardware and software sorting algorithms implemented in a SOC



Universidade de Aveiro Electrónica, Telecomunicações e Informática, 2017

Vítor Manuel
dos Santos Marques

Desempenho de algoritmos de ordenação em *hardware* e *software* implementados em SoC

Performance of hardware and software sorting algorithms implemented in a SOC

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Professor Doutor Valeri Skliarov, Professor Catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e da Professora Doutora Iouliia Skliarova, Professora Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

the jury

president

Tomás António Mendes Oliveira e Silva
Professor Associado da Universidade de Aveiro

examiners committee

Valeri Skliarov
Professor Catedrático da Universidade de Aveiro (orientador)

António José Duarte Araújo
Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto

acknowledgements

Primeiro de tudo quero agradecer aos meus pais.

Quero agradecer aos meus orientadores Prof. Doutor Valeri Skliarov e Prof. Doutora Iouliia Skliarova pelo acompanhamento e rigor exigido.

Quero agradecer a todos os meninos e a todas as meninas que me acompanharam neste longo percurso e pelas boas memórias.

Palavras-chave FPGA, Field-Programmable Gate Array, APSoC, Hardware e Software Co-design, Ordenação de dados, Algoritmos Paralelos

Resumo As Field Programmable Gate Arrays (FPGAs) foram inventadas em 1985 pela Xilinx. A sua natureza reconfiguratória permite que sejam utilizadas em várias áreas das tecnologias de informação.

Este trabalho tem como objectivo estudar o uso desta tecnologia como alternativa aos métodos tradicionais de processamento de dados, nomeadamente a ordenação. A solução proposta baseia-se na reutilização de recursos para combater as conhecidas limitações deste tipo de tecnologia.

Keywords FPGA, Field-Programmable Gate Array, APSOC, Hardware and Software Co-design, Data Sorting, Parallel Algorithms

Abstract Field Programmable Gate Arrays (FPGAs) were invented by Xilinx in 1985. Their reconfigurable nature allows to use them in multiple areas of Information Technologies.

This project aims to study this technology to be an alternative to traditional data processing methods, namely sorting. The proposed solution is based on the principle of reusing resources to counter this technology's known resources limitations.

Contents

List of Figures

List of Tables

Acronyms

APSoC All Programmable System-on Chip.

ASIC Application-Specific Integrated Circuit.

CPU Central Processing Unit.

DoD Department of Defense.

FPGA Field Programmable Gate Array.

FPU Floating Point Unit.

HDL Hardware Description Language.

IC Integrated Circuit.

IDA Institute for Defense Analysis.

IDE Integrated Development Environment.

IEEE Institute of Electrical and Electronics Engineers.

IP Intellectual Property.

PL Programmable Logic.

PLD Programmable Logic Device.

PROM Programmable Read-Only Memory.

PS Processing System.

RAM Random Access Memory.

RISC Reduced Instruction Set Computer.

SDK Software Development Kit.

USB Universal Serial Bus.

VGA Video Graphics Array.

VHDL VHSIC Hardware Description Language.

VHSIC Very High Speed Integrated Circuit.

Chapter 1

Introduction

During the decades 1960s and 1970s the Department of Defense (DoD) had many contractors that designed Integrated Circuit (IC). Each contractor had its own documentation, designs and way of work which made work together and share or reuse designs among different companies a daunting task.

To put an end to this diversity and to improve the performance of silicon based technology the DoD decided to fund the Very High Speed Integrated Circuit (VHSIC) program, which goal was to create a standard for Hardware Description Language (HDL) [1, 2, 3, 4]. That way there would be a design and documentation standard that allows reuse designs and clarifies processes. That HDL should also allow circuit simulation.

In 1981, Institute for Defense Analysis (IDA) held a workshop in Woods Hole, Massachusetts, to discuss the creation of a HDL standard [4, 5]. From that workshop a list of requirements came out to create VHSIC Hardware Description Language (VHDL).

In July 1983, it was attributed the contract to develop the new HDL and respective supporting software.

The first public available version of VHDL was released in 1985 [2, 3].

In 1986 Institute of Electrical and Electronics Engineers (IEEE) got interested in standardizing VHDL, after some modifications IEEE released its own standard, IEEE Standard 1076 [1, 2, 3, 4].

Currently, the DoD requires all Application-Specific Integrated Circuit (ASIC) delivered by contractors to be described in VHDL [1, 2].

At the same time the devices used in the industry were Programmable Read-Only Memory (PROM) and Programmable Logic Device (PLD) which are devices that can be programmed wherever the device is going to be used or during manufacture, but that can only be done once. They are hard-wired between logic gates which is a huge limitation. Engineering and developing new products is a process prone to errors, it is a trial and error process which can result in many chips being trashed. That fact lead to the idea of creating something that can be programmed multiple times without any limitations.

Ultimately, it lead to the creation of Field Programmable Gate Array (FPGA) which are devices which logic components that can be programmed multiple times at any place, be it at the factory during the manufacturing process or in the field during deploying and even years after being deployed.

In 1985 Xilinx released the first commercial available FPGA [6].

Since 1985, manufacturing processes evolved considerably. In fact it enabled the production

of advanced FPGAs broadening their field of use. Their field of applications is so wide and broad that they can be found in all kind of devices, from high performance computing such as servers to aerospace and defense such as aircraft and missiles systems. It should be noted that FPGA manufacturing processes also evolved. Today, it is possible to manufacture thinner slices of silicate which greatly improve the density of logic gates even reaching millions of logic gates [7]. Moreover, it is possible to have several peripherals such as Random Access Memory (RAM) or Universal Serial Bus (USB) port communication which makes FPGAs even more useful in a wider range of applications [8, 9].

Recent developments of FPGA, enable manufacturers to incorporate multi-core processors and programmable logic which enables both the capabilities of software and hardware on the same chip, making the so called All Programmable System-on Chip (APSoC). These kind of chips expand the field of applications even more because it allows engineers to build systems with the usual Programmable Logic (PL) that communicates with a Processing System (PS).

The fact that manufacturers can make FPGAs at low costs allows it to be a cost-effective alternative to ASIC which are tailored to a specific task and the process of designing and manufacturing that kind of chips is usually very expensive and take a long time-to-market [1, 10].

1.1 Motivation and Objectives

Nowadays computers and smart phones are very common, one third of the world's population owns a smartphone[11]. People use these devices for a large number of different activities such as on-line shopping, networking or social networks. All those activities generate huge amounts of data, however, that data has to be processed so it can be turned into something useful. In various data analysis projects sorting is a very common way of processing data [12]. FPGAs based hardware accelerators are being more commonly used for data processing[13].

The time it takes to sort a data set is directly related to its size, despite the existence of many sorting algorithms. This happens because each one of them performs in a different way and complexity. In fact, not every one of them has the same performance. There are some algorithms with a $O(n \log n)$ complexity, [14], such as heap sort, and sometimes even worse, $O(n^2)$ such as selection sort [14]. As previously said, data analysis using sorting methods is very important and done frequently so it makes sense that the processing time improves and the overall data throughput increases.

Central Processing Unit (CPU) work on high frequencies, usually in GHz, and single core CPU can only execute one instruction at a time. Taking into account that one comparison is deconstructed in multiple instructions it will turn a few lines of C/C++ code into multiple assembly instructions.

As far as FPGAs go, their frequency is slower, around 100s MHz, but they can be designed to execute multiple operations simultaneously due to parallelism.

It will be a case of executing one instruction at a time at a high speed against broad parallelism at a lower speed[15, 16].

The objective of this dissertation is to see which method has a greatest throughput, i.e., to compare the performance between hardware based accelerators and pure software based sorting algorithms.

As far as hardware goes this task will use the fundamentals of sorting networks.

1.2 Prototyping Systems

The prototyping board that will be used to develop the solution is the Xilinx Nexys4-DDR.

The FPGA is a member of the Artix-7 family, a low-cost and low power consumption FPGA manufactured by Xilinx[17].

This FPGA allows programmers to execute their designs written in a hardware description language such as VHDL or Verilog or in C/C++ and run them in the MicroBlaze soft microprocessor.

Programmers can design their project, write the code in a hardware description language such as VHDL or Verilog and run a simulation to see how the logic device will perform and make changes according to the desired results.

There is also the possibility of creating a soft microprocessor, named MicroBlaze, and use it as an ordinary micro controller. This model includes a 128 MiB DDR2 SDRAM memory which makes possible to store programs.

The MicroBlaze soft-core microprocessor designed for FPGAs manufactured by Xilinx. It is a soft microprocessor which means it is not an actual microprocessor, it is a virtual microprocessor that is synthesized to Xilinx's FPGAs just like any VHDL design.

MicroBlaze is a microprocessor with a 32-bit Reduced Instruction Set Computer (RISC) architecture optimized to FPGAs. MicroBlaze also has various configurations, the microprocessor can be optimized in various ways, besides the default configuration it can be optimized to occupy the minimum area possible, or to the maximum possible speed. MicroBlaze also can include a Floating Point Unit (FPU) and RAM among other options. With such a wide range of options MicroBlaze is highly re-configurable and as such it is very useful in a wide range of areas such as medical industry, communications and automotive areas.

It is also possible to use Intellectual Property (IP) Cores in conjunction with MicroBlaze, that way it is possible to create projects with PL and a PS, namely MicroBlaze microprocessor, with this it is possible to create more complex applications that can be used in more broad fields [18, 19].

The main components of the Nexys4-DDR are the following [20]:

- Xilinx Artix-7 FPGA XC7A100T-1CSG324C
- 128 MiB DDR2 SDRAM memory
- USB-UART Bridge
- 12-bit VGA output
- 3-axis accelerometer
- 128 MiB DDR2
- Pmod for XADC signals
- 16 user switches
- 16 user LEDs
- Two tri-color LEDs
- PWM audio output

- Temperature sensor
- Serial Flash
- Digilent USB-JTAG port for FPGA programming and communication
- Two 4-digit 7-segment displays
- Micro SD card connector
- PDM microphone
- 10/100 Ethernet PHY
- Four Pmod ports
- USB HID Host for mice, keyboards and memory sticks

In figure 1.1 it is possible to see the prototyping board used.

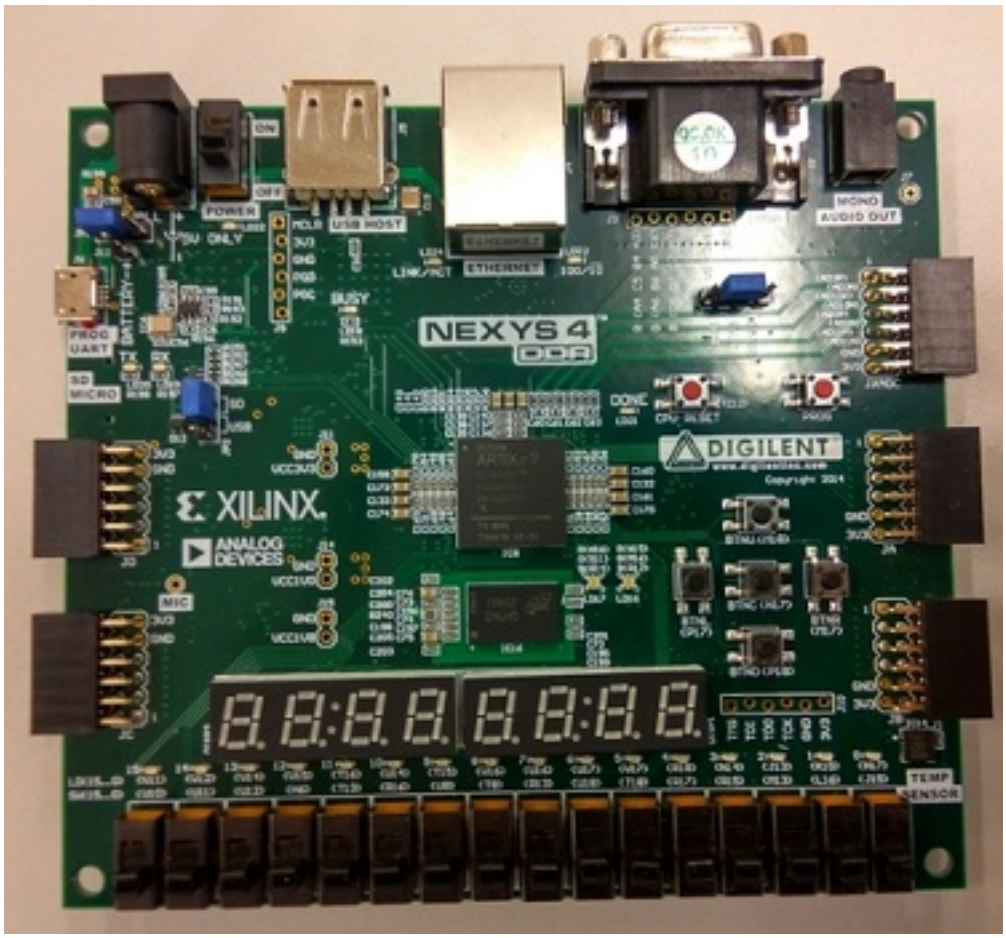


Figure 1.1: Simplified layout of Nexys4-DDR

1.3 Development Tools

The Xilinx Design Tools is a software suite released by Xilinx and it includes software to develop both hardware and software to be used on Xilinx boards. It is composed of two programs: Vivado and Xilinx SDK. Vivado is the software used to develop hardware, it allows users to design and create hardware using VHDL (or Verilog) or IP cores, synthesize, generate a bitstream and send it to the board. Xilinx SDK is an Integrated Development Environment (IDE) based on Eclipse and is used to write C/C++ code to run in the MicroBlaze soft microprocessor.

1.3.1 Vivado

Vivado is an IDE created by Xilinx to be used in the development of FPGA based solutions. It allows developers to create, code, simulate and synthesize designs to be used on Xilinx boards.

The software allows the creation of designs for specific Xilinx boards with the user choosing the specific models he wants to develop to.

This IDE is a fully fledged one, it is equipped with the usual tools present in the majority of software used by professionals to produce good quality code, and it has roughly the same layout of the most popular IDEs which makes life easier for those who are familiar with other IDEs. It has a code editor to allow users to write their code, a source viewer so that users can observe the project hierarchy (project structure and libraries used). The software also has debug features and a simulator so users can make simulations of their design to prevent errors and predict the design operation. Finally, it has the option to create the bitstream of the project so the user can program the board.

To create projects Vivado gives users more than one option, users may create their structures using a hardware design language such as VHDL or Verilog. It also gives the option of creating hardware blocks and use them in conjunction with Xilinx's own, known as IP Cores, to create more complex projects, including the soft embedded processor MicroBlaze.

In chapter 3 it will be shown in more detail how to create the sorting device.

In chapter 4 it will be shown more details about the MicroBlaze processor and used sorting algorithm.

1.3.2 SDK

The Xilinx Design Tools suite also includes a Software Development Kit (SDK).

The SDK is the recommended software-centric design environment to be used to build software solutions to run on Xilinx's soft embedded processors, namely MicroBlaze, and PowerPC [21].

The SDK is based on Eclipse, an open source IDE, which will be used to write C/C++ programs and comes with the GNU Software Development Tools to compile and debug programs for the previously referred platforms [22].

This program comes with a variety of options to manage the communication between the host computer and the development board. Among those options we can find:

- connection configuration (serial or via UART, bit rate)
- send the program to the FPGA

- using the SDK display the program output
- keyboard input
- debug purposes

In chapter 4 it will be shown the construction of a MicroBlaze processor and the sorting software.

1.4 Outline

The first chapter is an introduction to this thesis theme. The goal is to explain to the readers why it is important to study and research in this area. There is a small explanation about the platforms used and software required.

The second chapter is the state of the art. In this chapter it will be explained what are sorting networks and some examples will be studied in detail.

In the third chapter the development of a hardware based solution will be detailed. The proposed sorting network will be presented and discussed.

Chapter four is all about the software based solution. In this chapter it will be explained what are soft microprocessors and the requirements to synthesize a MicroBlaze soft processor. The software solution will be explained in this chapter.

The fifth chapter contains all tests done in order to compare both hardware and software-based solutions.

Finally, the last chapter is the conclusion. It contains the conclusions about all work done and final thoughts.

Chapter 2

State of the art

In general sorting networks are made of parallel wires and comparator modules connecting pairs of wires. Each wire holds an input value which will move towards the network's end. Those modules may swap their values if and if only they are not correctly sorted. As the values travel through the network the bigger values will "sink" while the smaller ones will "float". The set of comparator modules must be able to correctly sort any input set, no matter its order.

Figure 2.1 shows an example of a simple sorting network, in this specific case it a bubble sort.

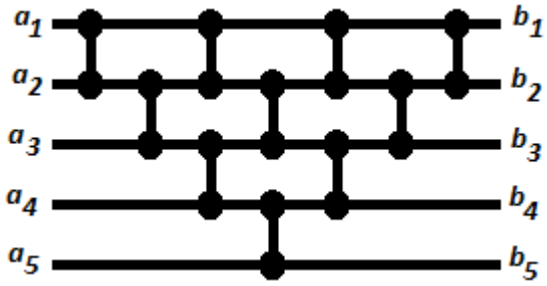


Figure 2.1: Sorting network - Bubble sort

Figure 2.2 represents both a comparator and a swapper, their behavior can be described in VHDL as the following:

```
MaxValue <= A when A >= B else B;
MinValue <= B when A >= B else A;
```

To be able to compare different sorting algorithms it is important to have some metrics, such as complexity or depth.

Length is the number comparators in a network.

Depth is the biggest quantity of comparators any input value has to go through to reach the network's end. Since we are dealing with hardware it's easy to parallelize some comparator modules and assuming all modules have the same time delay it can be presumed that a network's depth is the same as the number of steps it takes to execute.

It is possible to build optimal networks, networks with either optimal depth or optimal length, but in general it's not possible to find a network with both optimal length and optimal

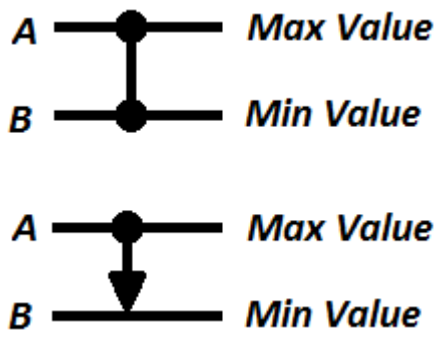


Figure 2.2: Comparator/Swapper

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14
length	0	1	3	5	9	12	16	19	25	29	35	39	45	51
depth	0	1	3	3	5	5	6	6	7	7	8	8	9	9

Table 2.1: Optimal length and depth values for sorting networks with N inputs [23]

depth [23]. In cases where hardware has limited resources it's interesting to build networks with as few comparators as possible saving resources. There are also situations where parallelism is the number one priority and so the depth of designed networks should be as low as possible. Table 2.1 shows the quantity of comparators to achieve optimal depth and optimal length for a given number of input items.

Some of the fastest and most used sorting networks are Batcher odd–even mergesort and bitonic merge sorting network [24, 25]. For those reasons those two networks will be explained in more detail in the following sections.

2.1 Batcher odd–even mergesort

Batcher odd-even mergesort was invented by Ken Batcher in 1968. The basic idea of this networks consists in merging two sorted subsets with the same number of inputs. Despite being named mergesort it is not data-dependent, it will always make the same quantity of comparisons despite the numbers to be merged. This makes it possible to transform it in a sorting network [25, 26].

The fundamental idea behind this algorithm is sorting two subsets and then merging them. Merging is done by sorting even indexed items and odd indexed items from both sets. After that it is only necessary to make a comparison per pair of items for the set to be completely sorted.

To make it simple to understand from here on it will be assumed that each set of items has N elements, $N = 2^p$.

Let's analyze the following situation. The following set of numbers must be sorted [27].

3 7 6 8 5 1 4 2

The first step consists in sorting each half of the set, the result is the following:

3 6 7 8 1 2 4 5

Next we proceed to sort odd indexed elements (3 7 1 4) and even indexed elements (6 8 2 5), the result is the following:

N	length	depth
2	1	1
4	5	3
8	19	6
16	63	10
32	191	15
64	543	21
128	1471	28
256	3839	36
512	9727	45
1024	24063	55
2^{20}	1×10^8	210
2^{30}	2.34×10^{11}	465

Table 2.2: Length and depth values for odd-even merge networks with N inputs

1 2 3 5 4 6 7 8

It easy to verify that the set is almost completely sorted. To finalize it is necessary to make comparisons of the items in indexes (2 and 3), (4 and 5) (6 and 7). And so the set is finally sorted.

1 2 3 4 5 6 7 8

According to [12, 28] it is possible to calculate length and depth of a Batcher odd–even mergesort for a set of any given size. As previously mentioned it will assumed the size of the input set is 2^p. The formula to calculate the network's length, i.e. the number of comparators in a networks is: $C(N = 2^p) = (p^2 - p + 4) \times 2^{p-2} - 1$ [25]. The formula to calculate the network's depth, i.e. the number of steps in a networks is: $D(N = 2^p) = p \times (p + 1)/2$ [25].

The length of this kind of networks is $O(N \log^2(N))$ [25]. The depth of this kind of networks is $O(\log^2(N))$ [25].

Table 2.2 shows the depth and length of Batcher odd–even mergesort networks for a given number of input items.

Figure 2.3 represents the trivial case and a 4 inputs example.

Figure 2.4 represents an 8 inputs network. One can notice it starts by sorting each half (represented by numbers 1 and 2), followed by merging both sub sets (3 and 4) and finally some comparisons between adjacent keys (5).

2.2 Bitonic merge sorting network

Interestingly bitonic merge networks were also invented by Ken Batcher in 1968. This algorithm consists in sorting a bitonic sequence. Although its name contains the expression "mergesort" it is not data-dependent, it will always make the same quantity of comparison despite the numbers to be merged. This makes it possible to transform it in a sorting network [25, 29].

The basic idea behind this algorithm is the 0-1 principle, it means if a comparator network sorts every sequence of 0's and 1's correctly, then it sorts all sequences of arbitrary numbers correctly.

Any sequence made of exclusively of 0's and 1's is named 0-1 sequence.

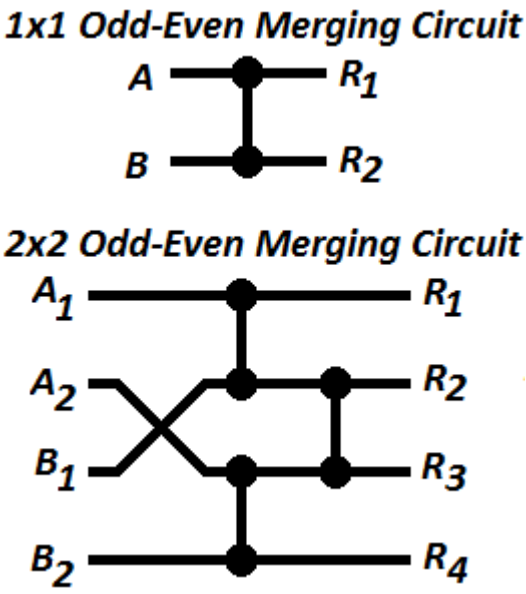


Figure 2.3: Odd-even networks - trivial examples

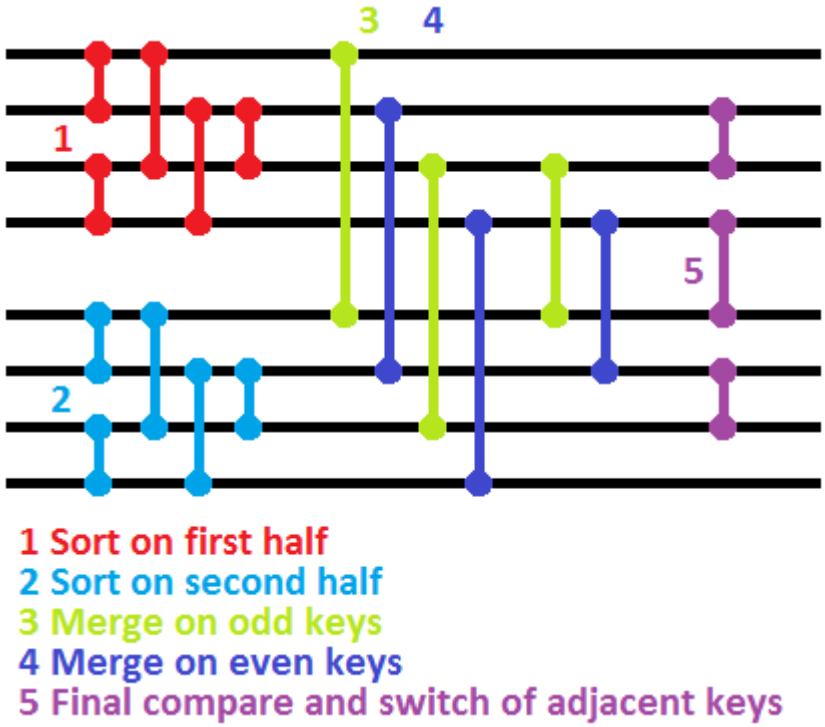


Figure 2.4: Eight inputs Batcher odd–even mergesort network

There is a particular case of 0-1 sequences which is very important to this sorting network, it is the so called bitonic sequences. Bitonic sequences are characterized by having at the most two transitions from 0 to 1 or 1 to 0, in other words, if there are sub sets with size $k, m \in \{1, ..., n\}$ so that

$$a_0, ..., a_{k-1} = 0, a_k, ..., a_{m-1} = 1, a_m, ..., a_{n-1} = 0 \text{ or}$$
$$a_0, ..., a_{k-1} = 1, a_k, ..., a_{m-1} = 0, a_m, ..., a_{n-1} = 1$$

Figure 2.5 represents all possible bitonic sequences. In figures 2.5, 2.7 and 2.8 white areas represent sequences of 0's and gray areas represent sequences of 1's.

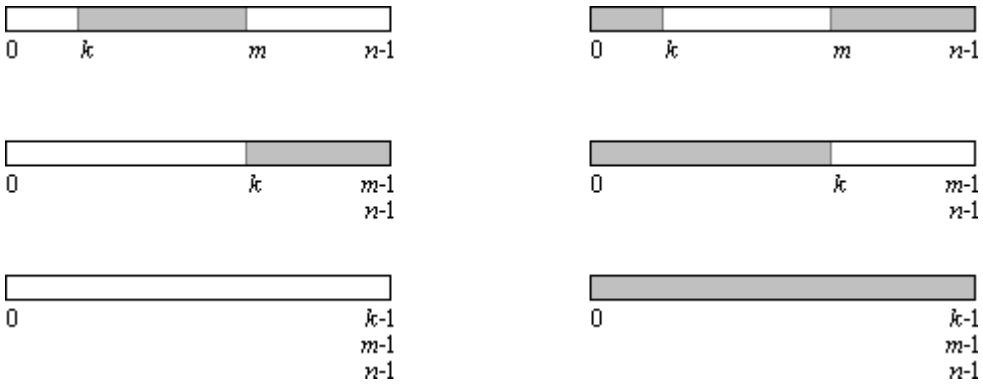


Figure 2.5: Bitonic sequences - examples [29]

Taking a comparator network B_n defined as:

$$B_n = [0 : n/2][1 : n/2 + 1]...[n/2 - 1 : n - 1]$$

Figure 2.6 is an example of this kind of networks.

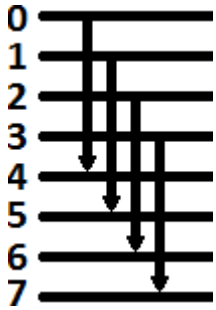


Figure 2.6: Comparator network B_8

By applying this comparator network to a bitonic 0-1 sequence A the result is

$$B_n(a) = b_0, ..., b_{n/2-1}c_0, ..., c_{n/2-1}$$

in which all b_i are less or equal to all c_j, which means

$$b_i <= c_j \text{ for all } i, j \in \{0, ..., n/2 - 1\}$$

also

$$b_0, ..., b_{n/2-1} \text{ is bitonic and}$$
$$c_0, ..., c_{n/2-1} \text{ is bitonic.}$$

To make it is easy to understand the result of this kind of comparator network one must write down the original sequence in two lines and apply the comparator network B_n. Figure 2.7 shows the result of applying a comparator network from figure 2.6 in various bitonic 0-1 sequences split in half.

Bitonic sorting networks use a divide and conquer strategy. It starts by dividing the sequence in half and then applying a bitonic sorter in each subset so that they become bitonic.

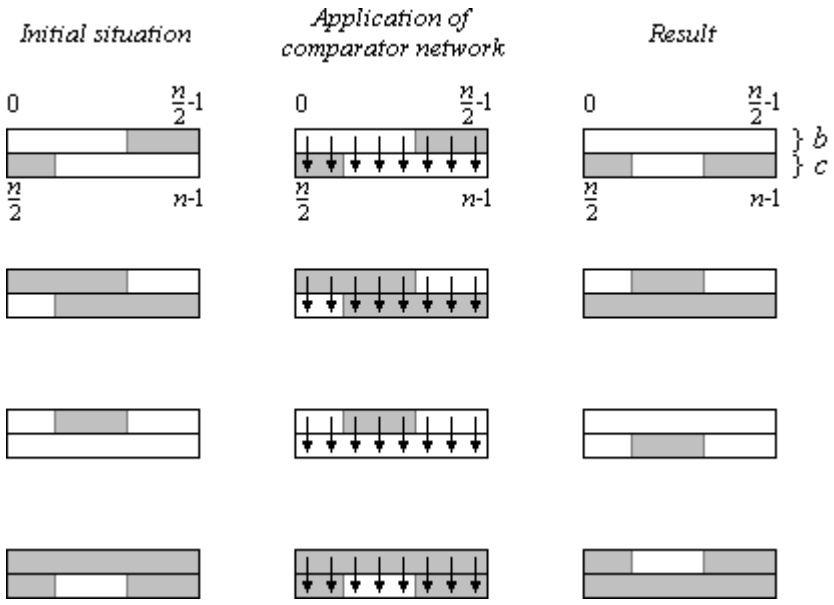


Figure 2.7: Application of comparator network B_n - examples[29]

After that a B_n comparator network is applied, this will result in all elements of the first line will be less or equal to all elements in the second one.

Bitonic merge networks are made of various processing blocks which change the input values to sort them.

The first step consists in splitting the original sequence in the middle and use a bitonic sort in both sub sets, this will end up moving all 1's towards the middle of the original sequence making it a bitonic 0-1 sequence. That is followed by the application of a comparator network B_n that makes both sub sets bitonic and all elements of the first sub set less or equal to the elements on the second one. Finally it applied a recursive bitonic sort in each subset.

As previously stated Bitonic Merge Sort implements a divide and conquer strategy. Figure 2.8 gives a general idea of all stages of that kind of strategy.

Figure 2.9 shows in great detail the implementation of an 8 inputs bitonic merge sort network and its stages in the divide and conquer strategy.

It is possible to calculate the length and depth of a Bitonic merge sorting network for a set of any given size.

As previously mentioned it will assumed the size of the input set is 2^p. The formula to calculate the network's length, i.e. the number of comparators in a networks is: $C(N = 2^p) = 2^{p-1} \times \sum_{i=1}^{p} i$. The formula to calculate the network's depth, i.e. the number of steps in a networks is: $D(N = 2^p) = \sum_{i=1}^{p} i$.

The length of this kind of networks is $O(N \log(N^2))$ [30]. The depth of this kind of networks is $O(\log(N^2))$ [30].

Table 2.3 shows the depth and length of Bitonic merge sorting networks for a given number of input items.

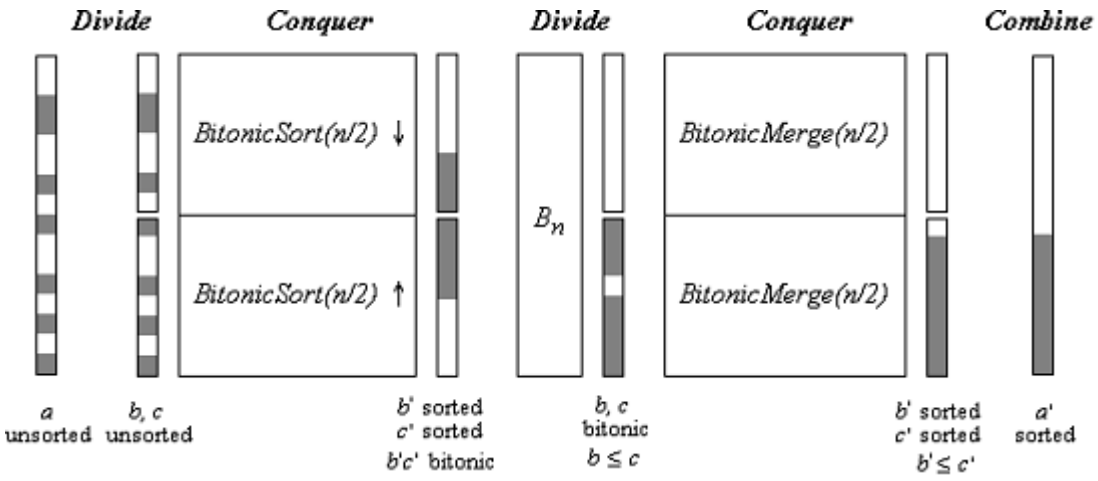


Figure 2.8: Generic Bitonic Merge Sort Network [29]

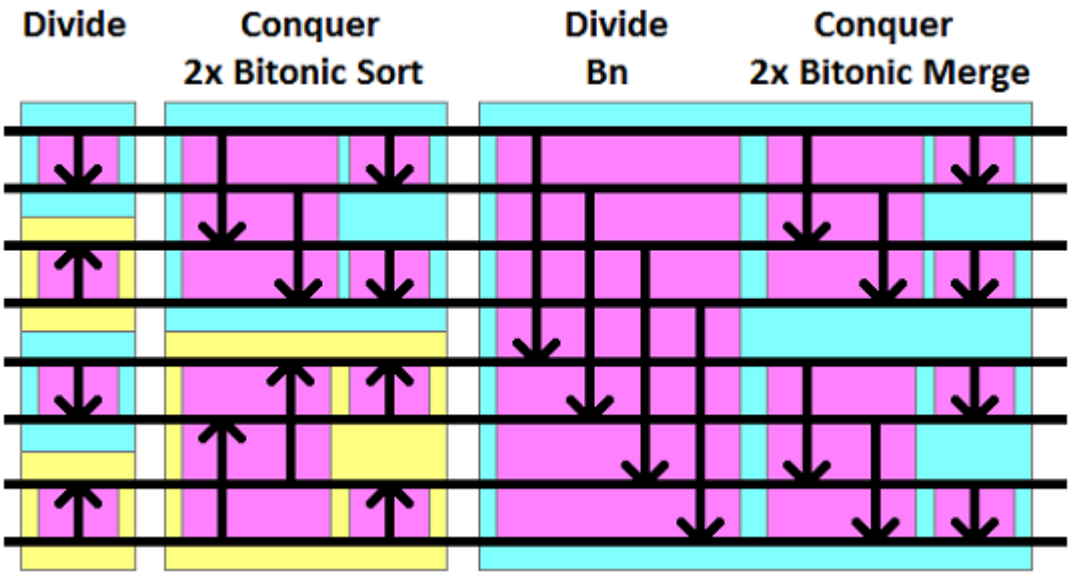


Figure 2.9: 8 Inputs Bitonic Merge Sort Network

N	**length**	**depth**
2	1	1
4	6	3
8	24	6
16	80	10
32	240	15
64	671	21
128	1792	28
256	4608	36
512	11520	45
1024	28160	55
2^{20}	1.1×10^8	210
2^{30}	2.49×10^{11}	465

Table 2.3: Length and depth values for Bitonic merge sorting networks with N inputs

Chapter 3

Hardware Development

The first part of this project consists in developing and synthesizing logic devices able to sort sets of numbers of any size. That process is going to be described in this chapter.

To sort sets of numbers we will have to build sorting networks big enough to handle any set of numbers. Unfortunately there a few problems with that, first, each comparator needs a certain quantity of hardware resources, slices, and second, FPGAs have limited resources. It prevents creating sorting networks that need a great number of comparators, or in other words, networks with big length[16]. That prevents us from synthesizing sorting networks able to sort large sets of numbers. Because their length is very high both Batcher odd–even mergesort and Bitonic merge sorting network are not very suited to be used to sort large sets.

Taking into account all these obstacles there is a possible solution. Let's study and build a solution that uses multiple times all those existing comparators, a solution that reuses comparators.

The current chapter is focused on studying, analysis and developing a solution able to sort a set of numbers based on such an architecture.

3.1 Sorting method description

As previously mentioned in section 1.1 every day large amounts of data are stored and it is quite common to use sort functions to extract useful knowledge from them[12]. If one decides to sort that data using sorting networks they will end up being gigantic.

Looking at tables 2.2 and 2.3 it is easy to see that for fairly small sets, i.e. $N = 32$, odd-even networks use 191 comparators and bitonic merge networks use 240 comparators. Each comparator used to sort two 32-bits numbers ($N = 2$, $M = 32$) will use at least 17 slices[31]. The board used in this project has 15 850 logic slices[32] which makes it impossible to synthesize +64 input items odd-even sorting networks. Even more advanced and expensive FPGAs, such as Xilinx's XC5VFX130T FPGA has the similar limitations[28, 33]. Currently in the market one of the most advanced FPGAs is Xilinx Virtex XC7V2000T and it has 305,400 slices[34], that means that it takes 5566 boards of this specific FPGA to sort a 2^{20} items set and not taking into account communications overhead.

To fight back the fact that it takes a huge amount of resources to build sorting networks a solution is to reuse existing comparators. The plan is to construct a network that is able to use the existing comparators iteratively until the set is sorted.

The design of this kind of networks must take into consideration factors of regularity and

scalability to enable construction of sorting networks able to sort sets of any size (any N). To boost comparator reuse it is convenient that the network is as regular as possible. This tries to ensure that the use of those comparators is increased as possible, that way it reduces the number of comparators doing few comparisons.

This way we will end up with network with low depth as possible, which means that it will not waste resources building a deep network and consequentially there will be more resources available to spend in each step. It enables constructing "wider" stages, in other words, stages able to make more comparisons simultaneously therefore sorting bigger sets of numbers.

This project has the goal of comparing both a hardware based solution against a software based solution so it makes sense to measure the circuit's throughput, i.e., the amount of sorted numbers per second. To increase the throughput we will have to augment the size of the sets to be sorted and as such the number of simultaneous operations. But taking into account that FPGAs have limited resources it is difficult to generate big networks which in turn makes difficult to sort big sets of numbers. On the other hand if we have to spend resources in the construction of a single stage, it will be possible to construct a "wider" one therefore enabling to sort bigger sets.

It is also necessary to take into account that PL usually functions at a lower frequency than PS. To try to ensure that the sorting network's throughput is greater than a software based solution a strategy of using large sets will be used to fight against overhead delays.

To design such solution an array of registers whose function is to store each iteration values will be used, it allows the iterative reuse of comparators. This sorting network is based on the circuit to discover the minimum and maximum values from [16, 33]. Figure 3.1 shows the design of such networks to sort an 8 inputs set.

This network is made of N M-bits registers, $N-1$ comparators and one multiplexer.

Those registers are used to store sorted values.

All comparators work simultaneously and they receive values originated from registers R_0 $\ldots R_{N-1}$. The network's output is directly connected to the registers array through feedback connections, except to register R_{N-1}. Initially all registers have a default value, in his case it will be 0 but it can be chosen another value deemed suitable.

Multiplexer M is directly connected to R_{N-1} and it is used to sequentially send values to the circuit or to signal the circuit we want to receive the sorted set.

Figure 3.2 shows the registers' content as the circuit receives the sequence 1) 28; 2) 14; 3) 37; 4) 65; 5) 11; 6) 14; 7) 19; 8) 71; 9) 0; 10) 69; 11) 14; 12) 41; 13) 71; 14) 22; 15) 70; 16) 7.

The moment the circuit has received an N item sequence it is ready to transfer them, as it can be seen in figure 3.3.

To receive the sorted set we must order the multiplexer M to send maximum values to the circuit. The maximum value is simply the maximum value any register $R_0 \ldots R_{N-1}$ can accommodate. Each time the circuit receives a maximum value it is moved up in the array, this transfers the minimum value to the R_{N-1} ready to be consumed.

Looking at figure 3.2 one can easily see each time a value is received the sets is getting partially sorted.

The set is completely sorted after the circuit stars to receive maximum values (clock cycle c3 in 3.3). In this particular case it happened in cycle c3 but meanwhile it has already transferred value during c1 and c2.

The formula to calculate the network's length, i.e. the number of comparators in a networks is: $C(N = 2^p) = 2^p - 1$. The formula to calculate the network's depth, i.e. the number of iterations is: $D(N = 2^p) = 2^p$.

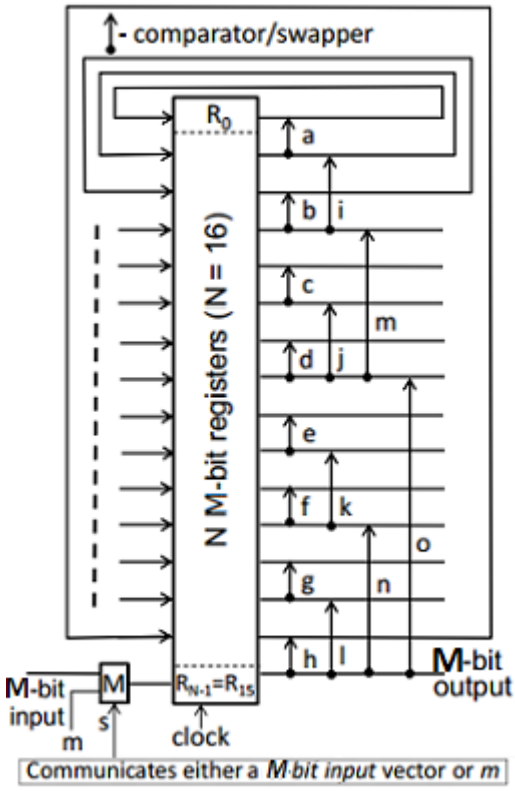


Figure 3.1: Proposed network [16]

The length of this kind of networks is $O(N)$. The depth of this kind of networks is $O(N)$.

Table 3.1 shows the depth and length of the proposed network for a given number of input items.

N	**length**	**depth**
2	1	2
4	3	4
8	7	8
16	15	16
32	31	32
64	63	64
128	127	128
256	255	256
512	511	512
1024	1023	1024
2^{20}	1.04×10^6	1.04×10^6
2^{30}	1.07×10^9	1.07×10^9

Table 3.1: Length and depth values for the proposed network with N inputs

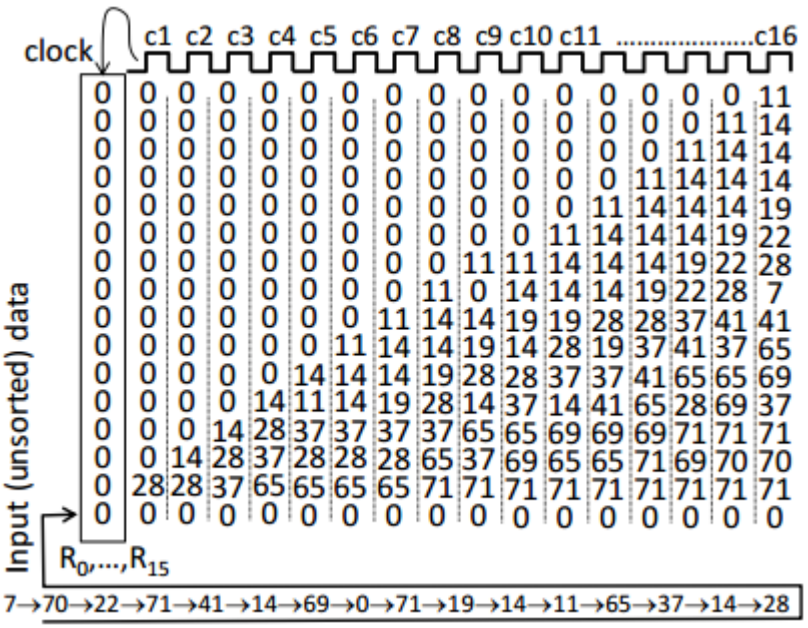


Figure 3.2: Data acquisition [16]

3.2 Circuit for discovering the minimum and maximum values

This section contains the VHDL code of the proposed circuit showed in figure 3.1 and the auxiliary modules used to display the sorted data in an external display.

It should be noted that it allows to sort sets of numbers within the limited resources of the given FPGA thus complying with the circuit's scalability requirements.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.std_logic_unsigned.all;
use work.Monitor.ALL;

entity NewIterativeSorter is
generic(N : integer:=256;
        M : integer:=32;
        p : integer:=8);
port (clk : in std_logic;
        reset : in std_logic;
        --select the page of the result
        sw : in std_logic_vector(2 downto 0);
        --ports related to the external monitor
        HSync : out std_logic;
        VSync : out std_logic;
        VGARed : out std_logic_vector(3 downto 0);
        VGAGreen : out std_logic_vector(3 downto 0);
```

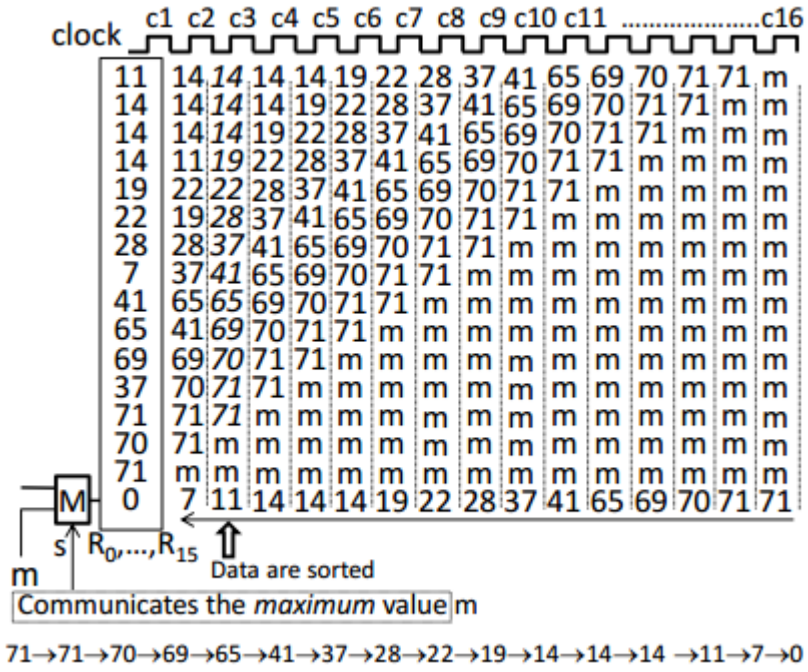


Figure 3.3: Data transmission [16]

```
        VGABlue : out std_logic_vector(3 downto 0);
        --leds to display the number of clock cycles
        LED : out std_logic_vector(15 downto 0));
end entity NewIterativeSorter;

architecture Behavioral of NewIterativeSorter is

signal MyAr1 : in_data_sort;
signal ind    : integer range 0 to N := 0;
signal ind_read    : integer range 0 to N := 0;
signal random_32bit : std_logic_vector(M-1 downto 0);

signal MyAr : in_data_sort;
signal MyAr2 : in_data_sort := (
x"00000050",x"00000040",x"00000060",x"11111111",
--
--values to be sorted
--
x"1900f211",x"9e0a421e",x"7ddea211",x"00000217");

signal sorting_completed : std_logic := '0';
--number of the page of the sorted array
signal page : integer range 0 to 7;
--time counting
```

```
signal tempo : integer range 0 to 65535 := 0;
signal tempo_out : integer range 0 to 65535 := 0;

begin
sorting: process (clk)
variable MyArV : in_data_sort;
variable tmp : std_logic_vector(M-1 downto 0);
begin
if rising_edge(clk) then
  if reset = '1' then
    --reset
    sorting_completed <= '0';
    ind <= 0;
    MyArV := (others => (others=>'0'));
    tempo <= 0;
  else
     if ((ind < N) and (sorting_completed = '0')) then
       ind <= ind + 1;
       MyArV(N-1) := MyAr2(ind);
       ind_read <= 0;
     else sorting_completed <= '1';
       if (ind_read < N) then
          MyAr(ind_read) <= MyArV(N-1);
          MyArV(N-1) := (others => '1');
          ind_read <= ind_read + 1;
          tempo_out <= tempo;
       end if;
     end if;

     --proposed circuit
     for k in 0 to p-1 loop
       for i in 0 to N/(2**(k+1))-1 loop
         if ( MyArV( 2**(k+1)*i+(2**k)-1 ) <
                              MyArV( 2**(k+1)*i+2**(k+1)-1 ) ) then
           tmp := MyArV( 2**(k+1)*i+(2**k)-1 );
           MyArV( 2**(k+1)*i+(2**k)-1 ) :=
                              MyArV( 2**(k+1)*i+2**(k+1)-1 );
           MyArV( (2**(k+1)*i+2**(k+1)-1) ) := tmp;
         end if;
       end loop;
     end loop;

     --time increment
     tempo <= tempo + 1;

  end if;
end if;
```

```
end process;

--copy sorted array to display array
process (page,MyAr)
begin
  for i in 0 to N-1 loop
    MyAr1(i) <= MyAr(((page*64)+i));
  end loop;
end process;

--VGA management
VGA_control : entity work.VGA
  port map (clk => clk,
            HSync => HSync,
            VSync => VSync,
            VGARed => VGARed,
            VGAGreen => VGAGreen,
            VGABlue => VGABlue,
            to_display => MyAr1);

--select page
page <= conv_integer(sw);

--output time to leds
LED <= conv_std_logic_vector (tempo_out, 16);

End Behavioral;
```

In the beginning it is possible to see that N is the number of items to be sorted and M is the number of bits of each value to be sorted.

The ports used include switches ports to be used to select which part of the sorted array should be displayed, leds to show the number of cycles it takes to sort the set and several ports related to the external display.

Inside the architecture it is possible to see that MyAr2 is the set with all items to be sorted, MyAr2's size must match N. Several auxiliary arrays and variables to hold the result and the measured time are also present.

As far as the sorting process goes, there is a reset state in which the time counter and the result array are reset. The other state implements the proposed circuit and counts the elapsed time. The moment the array is sorted it is copied to an auxiliary array.

Using the switches it is possible to select any section of the sorted array to be displayed in the external monitor.

There is also a controller that displays parts of the result in the monitor in the form of a matrix.

Finally, the leds are used to display the number of clock cycles it takes to sort the set.

3.3 Results check and consistency

As previously said, the focal point of this project is to compare two platforms (hardware and software) so, it is fundamental to have a way to compare both platforms. It will be done by comparing execution times.

This way it will be necessary to verify experimentally theoretical values. To achieve that we will count how many clock cycles it takes to sort a set and then display that value in the board's leds.

It is important to make the sorting process is correctly done. To ensure the sorting process is correctly done the final result is shown in an external monitor connected to the FPGA through its Video Graphics Array (VGA) port.

After sending the set to the circuit it is time to retrieve the sorted set. Each time the circuit receives a maximum value the smallest value sinks, it can be seen on figure 3.3. Thereafter that value is copied to an array whose function is to store the sorted set.

To display the final result it was decided to display it on an external monitor through a 64 item matrix, it can be seen in figure 3.5. To display the results of 64+ items tests it will be done through a page like system, the selection of each page will be done using the board's switches.

3.4 System up and running

In this section it will be shown images of the system running.

In figure 3.4 it is possible to see the ninth led is turned on, the leds represent the number of clock cycles it took to sort the set. The present current value was assessed during a 256 items test and the result in binary is 0000 0010 0000 0000 which in decimal represents 512.

Figure 3.4: Led turned on

Figure 3.5 shows the matrix filled with the sorted values.

Figure 3.5: Matrix displaying sorted set

Figure 3.6 is just a picture of the setup running (PC, FPGA and monitor).

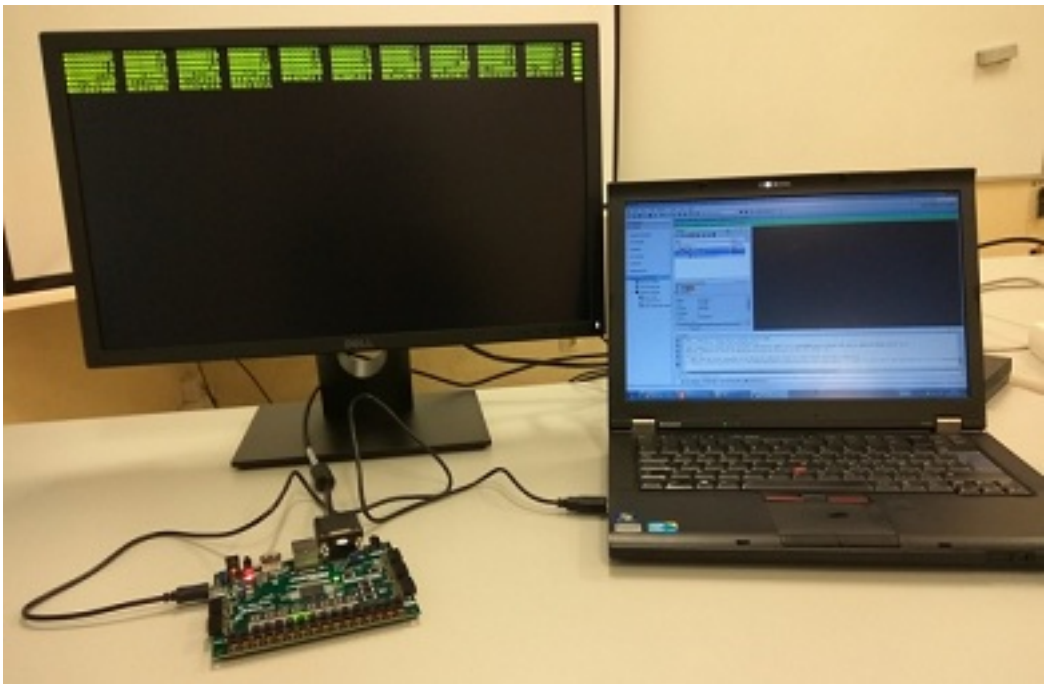


Figure 3.6: Full setup

Chapter 4

Software Development

The first part of this project consists in designing and synthesizing a programmable logic device to sort a set of numbers. That process is described in chapter 3.

The second part also consists in sorting a set of number using an ordinary C program.

To make things more interesting both parts will be executed in the same board, that means a board that is able to run programmable logic and a C program. FPGAs are able to run programmable logic but they do not have a micro processor to execute programs. There is workaround, a soft microprocessor will be used.

As a result of that we will have a hardware based sorter and software based one running on the same board, this will make the results much more interesting to discuss.

This chapter will be dedicated to designing and deploying an algorithm on the Nexys 4 DDR board. All processes related to the software will be described on this chapter, starting on the Vivado setup to be able to design MicroBlaze soft microprocessors, coding a C program until executing it.

4.1 MicroBlaze

MicroBlaze is a soft microprocessor by Xilinx.

Soft microprocessors are microprocessors fully implemented in programmable logic [19, 35].

Hard microprocessors are integrated circuits, that means they are physical devices that have to go through a design, test and production phases until they are commercially available. It is not possible to make any modification to microprocessors already manufactured, any change in its design means sending a new design to the factory to make new ones.

Soft microprocessors are completely implemented in programmable logic, they also go through design and test phases then they are deployed in FPGAs. Any modification in its design simply means redesigning, synthesize it and deploy in the FPGA, there is no production phase.

As there are many specifications of a single hard microprocessor it is possible to synthesize a soft microprocessor optimized accordingly to a specific requirement, such as frequency, performance, number of pipeline stages or minimal area used, that means that soft microprocessors can be tuned to specific requirements [19].

Table 4.1 shows multiple MicroBlaze specifications.

Pre-Defined Configuration	Brief Description
Minimum Area	Smallest possible MicroBlaze core, no caches, no debug
Maximum performance	Large caches, debug and execution unit
Maximum Frequency	Maximum achievable frequency. Small caches and no debug, with few execution units
Linux with MMU	Settings suitable to get high performance when running Linux with Memory Management Unit (MMU). Memory Management enabled, large caches and debug, and all execution units
Low-en Linux with MMU	Settings corresponding to the MicroBlaze Embedded Reference System. Provides suitable settings for Linux development on low-end systems. Memory Management enabled, small caches and debug
Typical	Settings giving a reasonable compromise between performance, area, and frequency. Suitable for standalone programs, and low-overhead kernels. Caches and debug enabled

Table 4.1: List of MicroBlaze possible specifications [19]

4.2 Vivado Setup

After installing Vivado there is a final step needed to be able to use DDR2/DDR3 memory blocks [36]. It is still necessary to get the Board Files to the Nexys 4 DDR. Those files are just XML files defining interfaces to the board. Those interfaces are Switches, Push Buttons, LEDs, USB-UART, DDR Memory, Ethernet, etc [36].

4.2.1 Vivado Configuration

Open up the window explorer and browse until you reach board_files in the directory Vivado is installed on, usually `C:\Xilinx\Vivado\2015.4\data\boards\board_files`.

Open your browser and download the Zip file:
`https://github.com/Digilent/vivado-boards/archive/master.zip`
Extract the contents of the Zip file and go to new/boards_files directory.
Copy its contents and paste them in board_files directory of Vivado.
Restart Vivado
Process completed

4.3 MicroBlaze implementation

In this section soft microprocessor MicroBlaze's design and implementation will be described.

A simple MicroBlaze microprocessor is simply not enough to complete the task, it is also necessary to use some extra peripherals. The first of those peripherals is RAM memory because the sorting algorithm requires more memory than the one MicroBlaze provides. The objective of this project is to compare the hardware based sorting algorithm with the software based

sorting algorithm and to do so there must exist some kind of metric to be able to compare, in this case it will be counting clock ticks it spent sorting.

As far as RAM memory goes it requires pre-configuration which is described in subsection 4.2.1.

The AXI Timer IP will be used to count the amount of ticks the qsort algorithm takes to sort a set.

4.4 Sorting Algorithm

As far as sorting algorithms go there were no constraints, by my advisor suggestion the qsort algorithm was used. qsort is a function available in standard C libraries [37].

qsort is based on quicksort and its average case has a $O(N \log(N))$ complexity and the worst case is $O(N^2)$[38].

The point of interest of this sorting algorithm is that it uses a user provided function to sort a data set. That way and given the implementation of the compare function it is possible to use qsort to sort any data type, including user defined types.

qsort function signature and parameters are the following [39]:

```
void qsort(void *base, size\_t nitems, size\_t size, int (*compare)
        (const void*, const void*));
```

Parameters:

- **base** - Pointer to the first element of the array to be sorted.
- **nitems** - Number of elements in the array pointed by base.
- **size** - Size in bytes of each array element.
- **compare** - Function that compares two elements.

Return value:

- This function does not return any value.

Relatively to compare function its signature and parameters are the following[37, 39]:

```
int compare(const void *p, const void *q);
```

Parameters:

- **p** - Pointer to the first element to be sorted.
- **q** - Pointer to the second element to be sorted.

Return value:

- $<$**0** The element pointed by p goes before the element pointed by q.
- **0** The element pointed by p is equivalent to the element pointed by q.
- $>$**0** The element pointed by p goes after the element pointed by q.

Both arguments are pointers to elements to be sorted. This it is possible to sort any data type, including user defined types, and order (increasing, decreasing or other).

The chosen implementation of the compare function is the following:

```
int compare(const void *p, const void *q)
{
    int x = *(const int *)p;
    int y = *(const int *)q;

    if (x < y)
        return -1;

    else if (x > y)
        return 1;

    return 0;
}
```

4.5 Software final version

The final version of the sorting program is the following:

```
#include <stdio.h>
#include "platform.h"
#include <stdlib.h>

#include "xil_types.h"
#include "xtmrctr.h"
#include "xparameters.h"
#include "xtmrctr_l.h"

//ELEMENTS is the number of elements to be sorted
#define ELEMENTS (10)

void print(char *str);
int compare(const void *p, const void *q);

int main()
{
    //initial configurations
    init_platform();
    int i;
    srand(0);
    rand();
    int my_array[ELEMENTS];

    //filling array with random values
    printf("CREATING...\n");
    for (i = 0; i < ELEMENTS; i++)
    {
```

```
        my_array[i] = rand();
    }

    //display original array
    printf("ORIGINAL\n");
    for(i = 0; i < ELEMENTS; i++)
    {
        printf("num[%d] = %d\n", i, my_array[i]);
    }

    //MicroBlaze timer setup
    XTmrCtr TmrCtrInstancePtr;
    unsigned int startTime, endTime, usedTime;
    XStatus Status = XTmrCtr_Initialize(&TmrCtrInstancePtr,
                        XPAR_AXI_TIMER_0_DEVICE_ID);
    if (Status != XST_SUCCESS)
    {
        printf("Problem with timer initialization. Exiting.\n");
        return 1;
    }
    //reset timer
    startTime = XTmrCtr_GetValue(&TmrCtrInstancePtr, 0);
    XTmrCtr_Start(&TmrCtrInstancePtr, 0);

    //sorting
    qsort(my_array, ELEMENTS, sizeof(int), compare);

    //measuring elapsed time and print
    XTmrCtr_Stop(&TmrCtrInstancePtr, 0);
    endTime = XTmrCtr_GetValue(&TmrCtrInstancePtr, 0);
    usedTime = endTime - startTime;
    printf("Number of ticks used: %d\n", usedTime);

    //display sorted array
    printf("SORTED\n");
    for(i = 0; i < ELEMENTS; i++)
    {
        printf("num[%d] = %d\n", i, my_array[i]);
    }

    cleanup_platform();
    return 0;
}

//auxiliary sorting function
int compare(const void *p, const void *q)
{
```

```
    int x = *(const int *)p;
    int y = *(const int *)q;

    if (x < y)
        return -1;

    else if (x > y)
        return 1;

    return 0;
}
```

The program creates an array of any desired size, set by the directive ELEMENTS, filled with random values.

Initially the program uses the rand function, seeded with the value 0, to fill the array with random values, after that the array is printed to see its contents.

After that MicroBlaze's timer is configured, the counter reset and started.

The qsort function is used to sort the array.

Following that, the timer is stopped and the elapsed time is measured and the value is printed.

The sorted array is printed to enable the user to see the process was correctly done.

Finally, some closing instructions are executed.

The compare function is used to sort the array.

4.6 System up and running

Figure 4.1 shows a simple example of the software based solution running. It is possible to see the execution flow and how many clock ticks it took to sort the set.

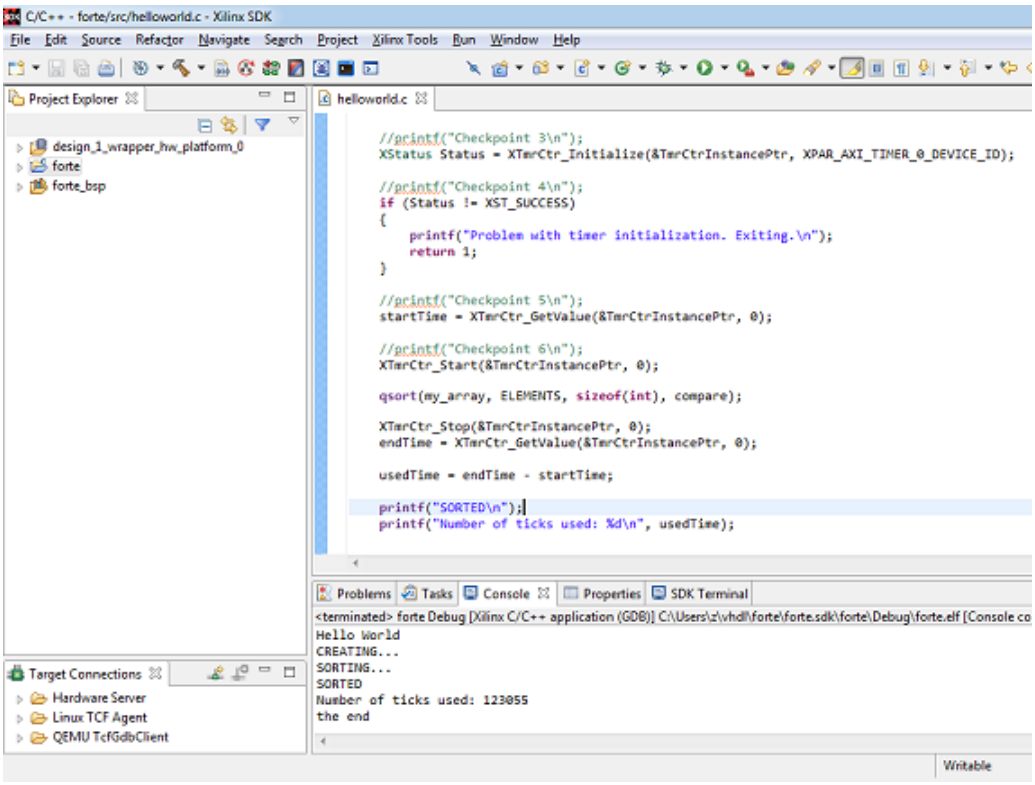


Figure 4.1: Execution of the software solution

Chapter 5

Tests

In this chapter all tests done and their results will be presented.

The focus of this project is to make a comparison between two platforms, hardware and software; it is very convenient to have similar tests for both platforms, this way it will be easier to compare them.

5.1 MicroBlaze tests

To test the qsort algorithm running on a MicroBlaze soft micro processor it was decided to make an average of 4 runs for each set whose size is $N = 2^p, p \geqslant 4$. Table 5.1 shows the average number of cycles taken to sort each set.

N	**qsort time**
16	18446
32	30298
64	60095
128	123006
256	251553
512	-

Table 5.1: qsort measured times

Figure 5.1 show the results from sorting a 256 items set and it shows the results were always different.

For unknown reasons the 512 items set test did not produce a result in a reasonable time.

Taking into account the time it took to sort a 256 items set the throughput of MicroBlaze is the following.

FPGA's frequency: 100 MHz = 10^8Hz

Ticks per set: 251553

Time per set:

$$\frac{251553}{10^8} = 0.00251533s$$

Sets per second:

$$\frac{1}{0.00251533} = 397.53sets/s$$

```
CREATING...
SORTING...
SORTED
Number of ticks used: 251603
the end

CREATING...
SORTING...
SORTED
Number of ticks used: 251557
the end

CREATING...
SORTING...
SORTED
Number of ticks used: 251598
the end

CREATING...
SORTING...
SORTED
Number of ticks used: 251629
the end
```

Figure 5.1: Execution of the software solution

Size of each 256 items set: 256×32 bits = 8Kb = 1KB

Throughput: $397.53 \times 1\text{KB} = 397.53$ KB/s

In figure 5.2 it is possible to see the quantity of the board's resources it took to synthesize the MicroBlaze soft micro processor. It is possible to see it takes less than half of the resources available.

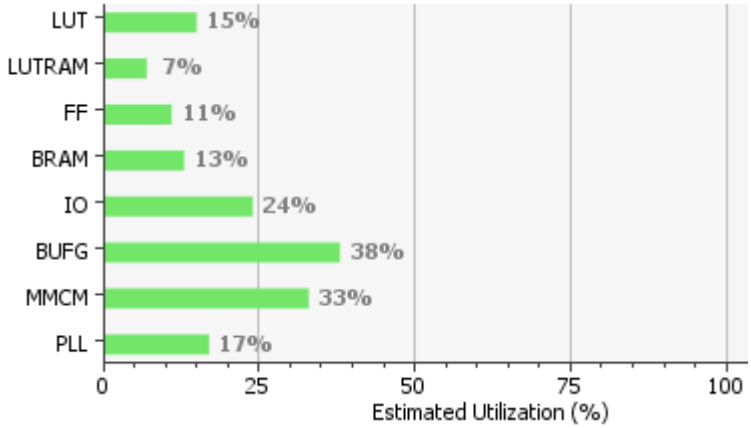


Figure 5.2: FPGA's resources usage for MicroBlaze implementation

5.2 Proposed circuit tests

To test the proposed circuit it was decided to make similar tests to the ones done in MicroBlaze. Table 5.2 shows how many clock cycles it took to sort certain sets.

N	Proposed circuit time
16	32
32	64
64	128
128	256
256	512

Table 5.2: Proposed circuit measured times

For each experience the used arrays were pre-built and then the circuit was synthesized. Figure 5.3 shows how many clock cycles it took to sort a 256 items set and it matches the theoretical value.

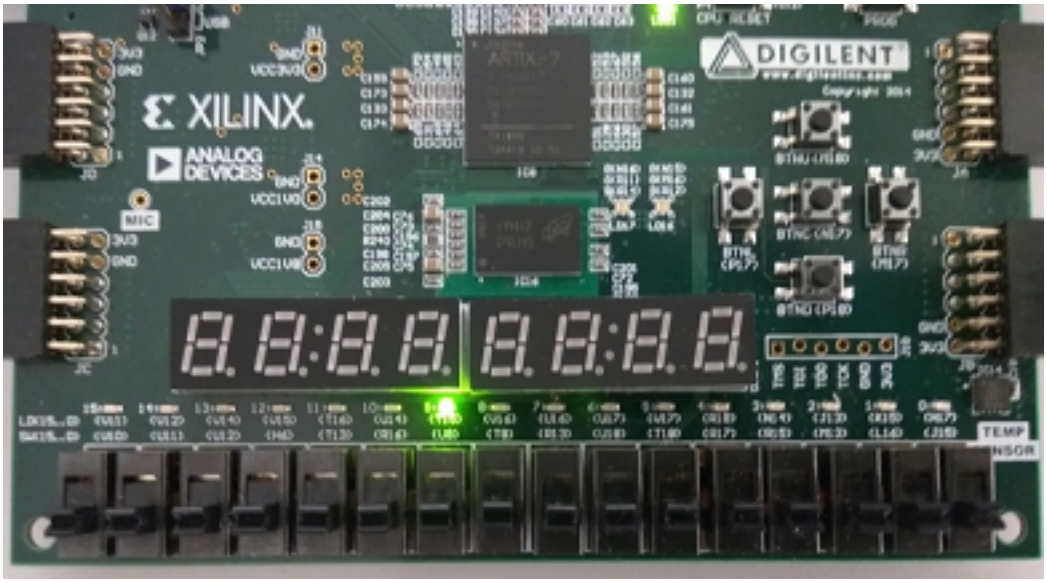


Figure 5.3: 256 items set time

Taking into account that all values obtained match with the theoretical values from table 3.1 and it was not possible to test a 512 items set in MicroBlaze it was not worth to make that test for the proposed circuit.

The values obtained for the 256 items set will be used to calculate the throughput.

FPGA's frequency: 100 MHz = 10^8Hz

Ticks per set: 512

Time per set:

$$\frac{512}{10^8} = 5.12 \times 10^{-6} s$$

Sets per second:

$$\frac{1}{5.12 \times 10^{-6}} = 195312.5 sets/s$$

Size of each 256 items set: 256 × 32 bits = 8Kb = 1KB

Throughput: 195312.5 × 1KB = 195312.5 KB/s = 190.7MB/s

In figure 5.4 it is possible to see the amount of FPGA's resources it took to synthesize a circuit to sort a 256 items set.

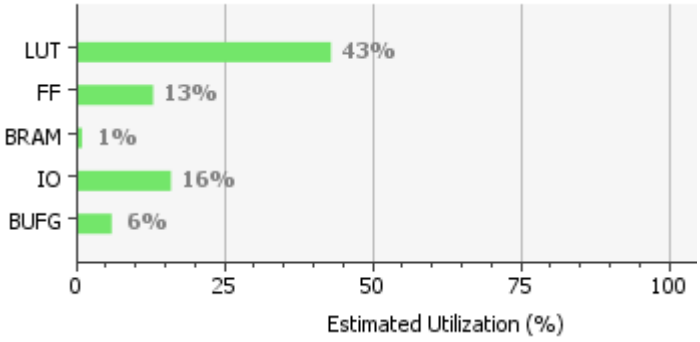


Figure 5.4: FPGA's resources usage for the proposed circuit implementation

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This dissertation describes all work done. It consists on analyzing and developing a circuit able to sort a set of numbers based on the principle of resources reuse. A MicroBlaze microprocessor was also synthesized and an ordinary software sorting algorithm was used to be able to compare both platforms.

Chapter one gives a brief introduction to FPGAs history, software tools used and the board used.

Following that a small introduction to the basics of sorting networks was given and then the most popular sorting networks were analyzed, including operating mode and complexity analysis.

After being more familiarized with the inner works of sorting networks the proposed circuit was presented and studied. The MicroBlaze soft microprocessor and the sorting algorithm were introduced.

After that it was time to make tests. Similar tests were performed in both platforms so that results are more conclusive. Running time results obtained from the proposed circuit were aligned with the analysis previously done.

As far as throughput goes, the proposed circuit largely supersedes the traditional software methods about 480 times. It is also important to evaluate the amount of resources used, in both cases less than half of all resources available were used. For unknown reasons it was not possible to tests the traditional software method for sets with 512 items.

Looking at the results one can see the sorting strategy using iteratively the same comparators is very effective.

Both the proposed circuit and the software were tested and their correct operation verified so it is safe to assume all targets were achieved.

Finally, it is important to refer that this area is continuously evolving because it enables the creation of specialized solutions in a multitude of fields.

6.2 Future Work

This dissertation may be used as foundation for future research. This work can be taken further by connecting a host computer to a FPGA through a PCI-Express bus, then the FPGA would be used as an external hardware accelerator. The host would send numbers to be sorted

in the FPGA and when that operation was completed, the FPGA would send the host a signal to indicate it is ready to send back the values sorted. Then it would be needed to redo the tests and compare to a host-run software.

Another take on this could be run this on a more advanced FPGA and compare the amount of resources as to this project.

Bibliography

[1] Kevin Shahill. VHDL for programmable logic Addison.Wesley Publishing, 1996

[2] Roger Lispett, Carl F. Schaefer, Cary Ussery. VHDL: Hardware description and design Kluwer Academic Publishers, 1993

[3] David Pellerin, Douglas Taylor. VHDL made easy! Prentice Hall, 1996, ISBN 0-13-650763-8

[4] James R. Armstrong, F. Gail Gray. VHDL design representation and synthesis Prentice Hall, 2000, ISBN 0-13-021670-4

[5] Zainalabedin Navabi. VHDL Analysis and Modeling of Digital Systems McGraw-Hill, 1993, ISBN 0-07-046472-3

[6] Funding Universe. `http://www.fundinguniverse.com/company-histories/xilinx-inc-history/` (last access May 16th, 2017)

[7] Clive Maxfield. The design warrior's guide to FPGAs Newnes, May 2004, ISBN 0750676043

[8] Xilinx, Inc. `https://www.xilinx.com/products/boards-and-kits/1-6olhwl.html` (last access May 16th, 2017)

[9] Xilinx, Inc. `https://www.xilinx.com/products/intellectual-property/axi_usb2_device.html` (last access May 16th, 2017)

[10] John V.Oldfield, Richard C. Dorf. Field-programmable gate arrays : reconfigurable logic for rapid prototyping and implementation if digital systems.Wiley-Interscience, January 23rd 1995, ISBN 978-0-471-55665-7

[11] Statista. `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/` (last access May 16th, 2017)

[12] D.E. Knuth. The Art of Computer Programming. Sorting and Searching, vol. III, Addison-Wesley (2011).

[13] V. Sklyarov, I. Skliarova, A. Barkalov, and L. Titarenko. Synthesis and Optimization of FPGA-based Systems, Springer, 2014.

[14] Thomas H. Cormen . . . [et al.]. Introduction to Algorithms - 3rd edition, The MIT Press, 2009.

[15] Xilinx, Inc. Zynq-7000 All Programmable SoC Technical Reference Manual, 2014. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf (last access May 16th, 2017)

[16] V. Sklyarov and I. Skliarova. Hardware Accelerators for Data Sort in All Programmable Systems-on-Chip

[17] Tech Focus Media. https://web.archive.org/web/20101123043600/http://www.techfocusmedia.net/fpgajournal/feature_articles/20100622-virtex/ (last access May 16th, 2017)

[18] Xilinx, Inc. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/mb_ref_guide.pdf (last access May 16th, 2017)

[19] Xilinx, Inc. http://www.xilinx.com/products/design-tools/microblaze.html (last access May 16th, 2017)

[20] Xilinx, Inc. https://reference.digilentinc.com/_media/nexys4-ddr/digilent_2015_nexys4ddr_ss.pdf (last access May 16th, 2017)

[21] Xilinx, Inc. http://www.xilinx.com/products/design-tools/platform.html (last access May 16th, 2017)

[22] Xilinx, Inc. http://www.xilinx.com/itp/xilinx10/help/platform_studio/ps_r_gst_whatsnew.htm (last access May 16th, 2017)

[23] Michael Codisha, Luís Cruz-Filipeb, Thorsten Ehlersc, Mike Mullerc, Peter Schneider-Kampb. Sorting Networks: to the End and Back Again, 4, 2015 https://arxiv.org/pdf/1507.01428v1.pdf (last access May 16th, 2017)

[24] V. Sklyarov and I. Skliarova, "High-performance implementation of regular and easily scalable sorting networks on an FPGA," Microprocessors and Microsystems, vol. 38, no. 5, 2014, pp. 470-484.

[25] K. E. Batcher. Sorting Networks and their Applications. Proc. AFIPS Spring Joint Comput. Conf., Vol. 32, 307-314 (1968)

[26] H.W. Lang, http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/oemen.htm (last access May 16th, 2017)

[27] Michel Goemans. Batcher's Algorithm, http://math.mit.edu/~shor/18.310/batcher.pdf (last access May 16th, 2017)

[28] R. Mueller, J. Teubner, G. Alonso. Sorting networks on FPGAs, Int. J. Very Large Data Bases 21 (1) (2012) 1–23

[29] H.W. Lang. http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm (last access May 16th, 2017)

[30] Greg Plaxton. Parallel Recursion: Batcher's Bitonic Sort, http://www.cs.utexas.edu/users/plaxton/c/337/05s/slides/ParallelRecursion-1.pdf (last access May 16th, 2017)

[31] V. Sklyarov and I. Skliarova, On-Chip Hardware Accelerators for Data Processing and Combinatorial Search

[32] Digilent Store. http://store.digilentinc.com/nexys-4-ddr-artix-7-fpga-trainer-board-recomm (last access May 16th, 2017)

[33] V. Sklyarov and I. Skliarova, "Fast regular circuits for network-based parallel data processing," Advances in Electrical and Computer Engineering, vol. 13, no. 4, 2013, pp. 47–50.

[34] Xilinx, Inc. "7 Series FPGAs Data Sheet: Overview"

[35] Xilinx, Inc. http://www.xilinx.com/ipcenter/processor_central/microblaze/doc/mb_faq.pdf (last access May 16th, 2017)

[36] Xilinx, Inc. https://reference.digilentinc.com/reference/software/vivado/board-files?redirect=1 (last access May 16th, 2017)

[37] GeeksForGeeks. http://www.geeksforgeeks.org/comparator-function-of-qsort-in-c/ (last access May 16th, 2017)

[38] JON L. BENTLEY, M. DOUGLAS McILROY. "Engineering a Sort Function", SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(11), 1249–1265 (NOVEMBER 1993)

[39] TutorialsPoint. https://www.tutorialspoint.com/c_standard_library/c_function_qsort.htm (last access May 16th, 2017)